Flexible VM Provisioning for Time-Sensitive Applications with Multiple Execution Options

Rehana Begam*, Hamidreza Moradi*, Wei Wang and Dakai Zhu
The University of Texas at San Antonio
San Antonio, TX, 78249
rehan.sheta@gmail.com, {Hamidreza.Moradi, Wei.Wang, Dakai.Zhu}@utsa.edu

***Abstract*—Several recent studies have investigated the virtual machine (VM) provisioning problem for requests with time constraints (deadlines) in cloud systems. These studies typically assumed that a request is associated with a single execution time when running on VMs with a given resource demand. In this paper, we consider modern applications that are normally implemented with generic frameworks that allow them to execute with various numbers of threads on VMs with different resource demands. For such applications, it is possible for the users to specify *multiple execution options (MEOs)* for a request where each execution option is represented by a certain number of VMs with some resources to run the application and its corresponding execution time. We investigate the problem of virtual machine provisioning for such time-sensitive requests with MEOs in resource-constrained clouds. By incorporating the MEOs of requests, we propose several novel and flexible VM provisioning schemes that carefully balance resource usage efficiency, input workloads and request deadlines with the objective of achieving higher resource utilization and system benefits. We evaluated the proposed MEO-aware schemes on various workloads with both benchmark requests and synthetic requests. The results show that our MEO-aware algorithms outperform the state-of-the-art schemes that consider only a single execution option of requests by serving up to 38% more requests and achieving up to 27% more benefits.**

***Index Terms*—Cloud systems; Resource allocation; Virtual machines; Deadlines; Multiple execution options;**

I. Introduction

Cloud computing platforms are rapidly emerging as the preferred option for hosting applications in many business contexts [7]. It enables on-demand and flexible provisioning of a shared pool of hardware resources (e.g., CPU, memory, disks and networks) to user applications. There is a myriad of technologies and services that are required for the effective operation of cloud computing systems. One of the most crucial services for resource efficiency of cloud computing is the effective resource provisioning strategy.

Undoubtedly resource provisioning (i.e., the selection and deployment of VMs of user jobs) is an important and challenging problem in cloud computing environments and has been intensively studied in prior research [3], [2], [31], [21], [6]. Although prior studies have investigated the VM provisioning problem for time-sensitive jobs (which have deadlines) on resource-constrained clouds, they usually assumed that a user request (job) has only one execution option (denoted by its execution time with a certain resource demand). However, modern applications, such as high-performance computing applications and machine-learning applications, are usually implemented with parallel frameworks, such as Pthreads, OpenMP, MPI and Tensorflow [25], [9], [12], [1]. These frameworks allow modern applications to executed with various numbers of threads on servers with different resources. Therefore, when submitting a request (job) to a cloud system, a user may have several options on how the application can be executed. Here, each option can be specified by a requirement of resources in terms of VM sizes and VM counts, as well as a unique estimated execution time of the application based on the resource demand.

When there are a range of execution options for executing an application, it becomes difficult and too rigid for the user to choose only one (or the best) execution option. More importantly, our previous experience in cloud VM provisioning has shown that one major cause of low system utilization is this rigid resource demand model, which limits the provisioner's ability to fully utilize the un-allocated resource fragments. Specifically, if a job is specified with only one execution option that demands more resources than any un-allocated resource fragments, such an option will prevent the job from being issued for execution, even if the job's implementation may allow it to use fewer numbers of threads and VMs. As shown in Section III, having *multiple execution options (MEOs)* for each job provides opportunities for more flexible VM provisioning.

Note that, simply adding more execution options does not guarantee improved system utilization and benefits. In fact, it may even lead to worse performance if improper options are chosen as most parallel applications do not have perfect linear speedups and normally experience lower resource efficiency for systems with more resources [26]. For instance, if a high-resource demand execution option is chosen for a job, it may block more resources for an extended period than low-resource demand option due to lower resource efficiency, which could prevent the resources from being allocated to other jobs and in turn lead to more jobs missing deadlines and lower system benefits. Consequently, to realize the full potential of multiple execution options of jobs, VM provisioning algorithms must be carefully designed to balance system utilization, resource efficiency, input workloads, job deadlines and benefits.

In this paper, we present two novel MEO-aware VM pro-

*The first two authors made the same contribution to this work. This work was supported in part by NSF grants CNS-1422709 and CCF-1617390.

visioning schemes named *MEO-Greedy* and *MEO-Adaptive*. Here, both schemes consider the MEOs of requests and prioritize them based on current system load, resource efficiency of execution options, deadlines and job benefits (bids). With the priorities, requests are provisioned with resources by either selecting the more efficient execution options to increase system utilization, or adapting (scaling up) the execution options gradually to maximize resource utilization with most efficient options. The proposed schemes were evaluated using a simulator for various workloads (e.g., overload and spiky scenarios) with requests of both benchmark and synthetic applications. The results show that our MEO-aware schemes outperform the state-of-the-art schemes that consider only a single execution option of jobs by serving up to 38% more jobs and achieving up to 27% more benefits.

The remainder of this paper is organized as follows. We discuss the closely related works in Section II. Section III presents system models and a motivation example. The MEO-aware VM provisioning schemes for time-sensitive applications are proposed in Section IV. Section V discusses the evaluation results and Section VI concludes the paper.

II. Closely Related Works

The problem of VM provisioning in cloud systems has been investigated from different points of view and many research studies have been conducted on resource allocation mechanisms [4], [34]. For instance, the provisioning techniques have been investigated to improve the performance of the user applications [20], [21], [14], to efficiently use cloud resources [35], [16], to minimize the user cost and maximize the revenue for cloud service providers [36], [11], [27], [28], to deliver services to the cloud users even in presence of failures [18], [17], to improve QoS parameters [30], [32], or to reduce power consumption [19], [15]. However, only very limited research has focused on time-sensitive applications with deadlines.

For jobs with timing-constraints, Li et al. introduced their DCloud resource allocator that leverages the (soft) deadlines of jobs in public clouds [22]. For deadline-constrained bag-of-tasks applications, a set of algorithms were proposed to cost-efficiently schedule them on a hybrid cloud [8]. Toosi et al. proposed a resource provisioning algorithm to support the deadline requirements of data-intensive applications in hybrid cloud environments by measuring the walkability index [29]. An architecture for coordinated dynamic provisioning and scheduling that is able to cost-effectively complete applications within their deadlines were studied for hybrid clouds in [5]. These studies did not consider jobs with hard deadlines and private clouds that normally have limited resources.

Le et al. compared the performance of several classic scheduling algorithms for jobs with deadlines in cloud systems, including first-come-first-serve, shortest job first and EDF algorithms [21]. A heuristic workflow scheduling algorithm was proposed that attempts to minimize the execution cost considering a user-defined deadline constraint [10]. Lim et al. focused on resource allocation and scheduling on clouds and clusters that process MapReduce jobs with SLAs [23]. Vecchiola et al. designed a deadline-driven resource manager for scientific applications running on hybrid clouds [31]. While these studies considered deadlines for time-sensitive jobs, they have assumed a rigid execution model where jobs have only a single execution option.

III. Preliminaries and Motivating Example

A. Cloud and VM Models

We consider a cloud system that consists of M heterogeneous computing nodes $\Pi = \{\mathbb{N}_1, \ldots, \mathbb{N}_M\}$. Each node has R types of resources $\Gamma = \{\mathbb{R}_1, \ldots, \mathbb{R}_R\}$ (such as CPU cores, memory and network bandwidth). The capacity vector of node $\mathbb{N}_p$ is denoted as $\mathbf{C}_p = (c_p^1, \ldots, c_p^R)$, where c_p^r represents the total capacity of resource $\mathbb{R}_r$ on node $\mathbb{N}_p$ (e.g., number of CPU cores). With heterogeneous nodes being considered, the capacity of one resource in different nodes can be different.

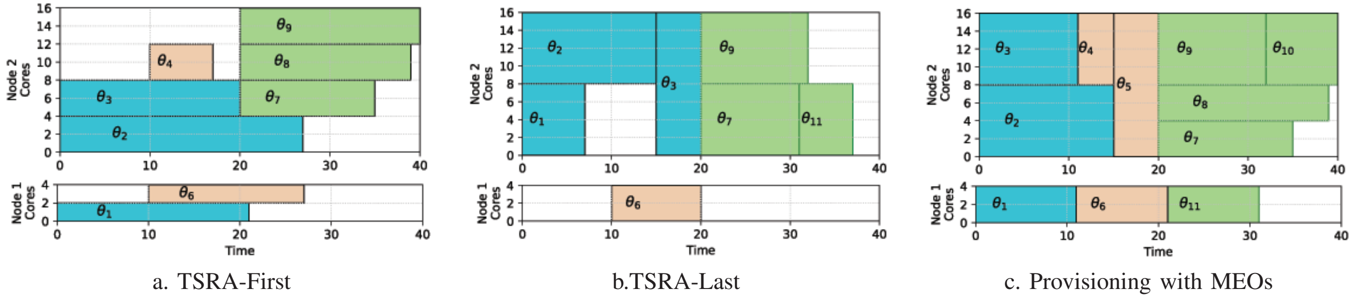
The computing resources in the cloud can be accessed by cloud users in the form of virtual machines (VMs). In this work, we consider V types of virtual machines $\{\mathbb{V}_1, \ldots, \mathbb{V}_V\}$ that have different resource requirements. For the VM type $\mathbb{V}_k$, its required resources can be denoted by a demand vector $\mathbf{W}_k = (w_k^1, \ldots, w_k^R)$, where w_k^r represents the required capacity (or amount) of resource $\mathbb{R}_r$.

B. Requests with Multiple Execution Options (MEOs)

To run a time-sensitive application, a cloud user needs to submit a *request* to the cloud system, which can be represented as a tuple $\theta_i = (a_i, d_i, b_i, E_i)$. Here, a_i denotes the request's arrival time, d_i defines the deadline by which the application needs to complete its execution, and b_i represents the bid (or benefit) that the system can achieve when the application completes its execution in time. Since an application may run with different number of threads on one or more VMs with different number of cores (and other resources), a cloud user can specify *multiple execution options (MEOs)* for running the application, which form the execution vector $\mathbf{E}_i = \langle\langle v_{i,1}, m_{i,1}, t_{i,1}\rangle, \langle v_{i,2}, m_{i,2}, t_{i,2}\rangle, \ldots, \langle v_{i,j}, m_{i,j}, t_{i,j}\rangle\rangle$.

Here, the kth option is denoted as $\langle v_{i,k}, m_{i,k}, t_{i,k}\rangle$, where $v_{i,k}$ ($\in [1, V]$) and $m_{i,k}$ represent the type and number of VMs needed to run the application, and $t_{i,k}$ denotes its (estimated) worst-case execution time on the deployed VMs. Note that, parallel applications normally take less time to run when more threads are used on systems with more cores (and other resources). Without loss of generality, it is assumed that the execution options are ordered in decreasing execution times where $t_{i,1} > t_{i,2} > \ldots > t_{i,j}$. That is, it takes the longest time to run the application with the first option that normally needs the least amount of resources. In comparison, the last (or *most-demanding*) option needs the most resources but the least amount of time to run the application.

Given that user requests for running some applications can arrive dynamically at anytime, we assume an *interval-based* processing strategy, where the length of an interval can be configurable (i.e., tens of minutes or a few hours). We further assume that there are n requests (including those



a. TSRA-First b.TSRA-Last c. Provisioning with MEOs

Fig. 1: Example: VM provisioning/schedules under different schemes.

arrived during the last interval and un-served ones from past intervals) for being processed in an interval with its beginning time at t_0, which form the request set $\Theta(t_0) = \{\theta_1, \dots, \theta_n\}$.

Problem Description: *For a given set of requests $\Theta(t_0)$ to process user's time-sensitive applications on a cloud system with M computing nodes $\Pi = \{\mathbb{N}_1, \dots, \mathbb{N}_M\}$, finding the best VM provisioning scheme such that the overall system bids (or benefit), which is defined as the aggregated bids of user requests that are served in time, is maximized.*

Different from public clouds where many VMs may overload a computing node and share its resources, to ensure the timely executions of time-sensitive applications, we assume dedicated resources are allocated to the VMs that are mapped to the same node simultaneously, which is similar to some recent studies [2], [24], [33]. That is, for each virtual CPU demanded by a VM, it will be allocated a physical CPU core from the computing node (which is the same for other resources). In addition, we consider only the capacity/demand of a resource (e.g., number of cores or amount of memory) without differentiating its actual model and specification (such as operating frequency or speed).

It is not hard to see that finding the optimal VM provisioning for the requests within an interval is NP-hard. In addition, the best VM provisioning for a single interval may not lead to the maximum overall system benefit when multiple intervals are considered. Therefore, in this work we focus on exploiting the MEOs of requests and designing efficient heuristics. In what follows, through a concrete example, we first illustrate how the flexibility of requests' MEOs can be exploited to improve system performance with more achieved benefits.

C. A Motivating Example

For simplicity, we consider a cloud system that has only two nodes $\mathbb{N}_1$ and $\mathbb{N}_2$ with 4 CPU cores, 8GB memory and 16 CPU cores, 32GB memory respectively. That is, $\mathbf{C}_1 = (4, 8)$ and $\mathbf{C}_2 = (16, 32)$. Three types of virtual machines $\mathbb{V}_1$, $\mathbb{V}_2$ and $\mathbb{V}_3$ are considered with their demand vectors as $\mathbf{W}_1 = (2, 4)$, $\mathbf{W}_2 = (4, 8)$ and $\mathbf{W}_3 = (8, 16)$ respectively.

The length of each interval is assumed to be 10 time units. There are 3, 3 and 5 requests that arrive at the beginning of the first three intervals, respectively, as shown in Table I. Here, each request has two or more execution options. Since most applications do not scale perfectly, for an option with more cores (and other resources), the execution time of the application with more threads does not decreases linearly, which actually leads to increased workload. For the requests in Table I, their least-demanding (most-demanding) options result in around 50%, 90% and 120% (160%, 180% and 280%) system loads for the three intervals, respectively.

TABLE I: Sample Requests to be processed

θ_i	(a_i, d_i, b_i, E_i)
θ_1	$(0, 26, 16, \langle\langle 1,1,21\rangle, \langle 2,1,11\rangle, \langle 3,1,7\rangle\rangle)$
θ_2	$(0, 29, 10, \langle\langle 2,1,27\rangle, \langle 3,1,14\rangle\rangle)$
θ_3	$(0, 24, 4, \langle\langle 2,1,20\rangle, \langle 2,2,11\rangle, \langle 2,4,6\rangle\rangle)$
θ_4	$(10, 20, 9, \langle\langle 2,1,7\rangle, \langle 2,2,4\rangle, \langle 2,4,3\rangle\rangle)$
θ_5	$(10, 24, 3, \langle\langle 3,1,9\rangle, \langle 3,2,5\rangle\rangle)$
θ_6	$(10, 37, 18, \langle\langle 1,1,17\rangle, \langle 1,2,10\rangle\rangle)$
θ_7	$(20, 36, 25, \langle\langle 2,1,15\rangle, \langle 2,2,11\rangle\rangle)$
θ_8	$(20, 42, 17, \langle\langle 2,1,19\rangle, \langle 3,2,7\rangle\rangle)$
θ_9	$(20, 44, 9, \langle\langle 2,1,20\rangle, \langle 2,2,12\rangle\rangle)$
θ_{10}	$(20, 41, 3, \langle\langle 2,2,8\rangle, \langle 2,4,5\rangle\rangle)$
θ_{11}	$(20, 36, 1, \langle\langle 2,1,10\rangle, \langle 2,2,6\rangle\rangle)$

In our prior work [2], we have studied the VM provisioning problem for time-sensitive applications with only one execution option and proposed the *Time-Sensitive Resource Allocation (TSRA)* scheme. Basically, TSRA prioritizes requests based on a factor that incorporates their resource demands, bids and deadlines and allocates their VMs to computing nodes accordingly. For the requests in Table I, by considering only their first options, Figure 1a shows their VM provisionings (in terms of CPU cores) under TSRA (denoted as *TSRA-First*).

Here, based on their TSRA factors [2], the requests in the first interval are ordered as: $\theta_1 \succ \theta_2 \succ \theta_3$, where request θ_1 is served first, followed by θ_2 and then θ_3. Since the requests are served with their first (least-demanding) options, half of the CPU cores are left unused in the first interval. Similarly, the requests in the second interval can be ordered as $\theta_6 \succ \theta_4 \succ \theta_5$ based on their TSRA factors. Here, after the requests θ_6 and θ_4 are served, the available resources are not enough to host the request θ_5's VM, which cannot be served in time and misses its deadline. Finally, the requests in the third interval are served in the order of $\theta_7 \succ \theta_8 \succ \theta_9 \succ \theta_{10} \succ \theta_{11}$. Again, the requests θ_{10} and θ_{11} cannot be served in time due to lack of resources and are discarded. In summary, TSRA-First discards 3 out of

11 requests and achieves 72% of the overall available benefits.

Instead of the first option, we can apply TSRA and focus on the last (most-demanding) option of the requests (denoted as *TSRA-Last*) and Figure 1b illustrates the VM provisionings for the requests. Here, we can see that, although TSRA-Last can utilize the resources relatively better in the first interval, the big chunks of demanded resources cause large fragmentation and actually lead to more requests missing their deadlines, where four requests θ_4, θ_5, θ_8 and θ_{10} are discarded. Here, TSRA-Last achieves only 63% of overall available benefits.

With the requests' MEOs being considered, a flexible VM provisioning scheme should adaptively select the most appropriate execution option for each request based on the available resources and the timing-constraints of the requests. As shown in Figure 1c, to better utilize the resources in the first interval, requests θ_1 and θ_3 are allocated with their second options, while the request θ_2 with its last (most-demanding) option. Similarly, by selecting the appropriate execution options for all other requests, all of them can be served in time and 100% of overall benefits are achieved. However, we want to point out that finding the appropriate execution option for each request is not trivial and the details of our proposed flexible MEO-aware VM provisioning heuristics are discussed next.

IV. Flexible MEO-aware VM Provisioning

A. Overview of MEO-TSRA

Due to the NP-hard nature of the task scheduling problem in general [13], heuristics are usually adopted in resource provisioning approaches in clouds. In this paper, we present two MEO-aware time-sensitive resource allocation (MEO-TSRA) heuristic schemes. These two heuristic schemes balance the needs of improving system utilization and resource efficiency to increase job deadline satisfactions and achieved benefits.

Both schemes have the same three-step overall structure, which is illustrated with the MEO-TSRA function in Algorithm 1. At the beginning of a new time interval T, the MEO-TSRA function is invoked with a heuristic scheme to allocate resources for T. The first step of MEO-TSRA is to collect current requests and resource availability (line 3 - 4). Both new requests arriving at T and old requests in the wait queue are collected. In the second step, the requests are ranked and prioritized based on the resource efficiency (scalability) of their execution options, their urgency (deadlines and execution times), benefits (bids) and resource demands (line 6). The exact prioritization algorithm is described in Section IV-B. In the last step, the prioritized requests and current resource availability are passed to either of the two heuristic schemes, MEO-Greedy or MEO-Adaptive, to allocate resources for interval T. These two heuristic schemes are described in Section IV-C and IV-D in details.

B. Prioritization of User Requests

The goal of request prioritization is to determine the requests that should be executed earlier than the other requests. Intuitively, requests with higher benefits and higher urgency should be executed earlier. Furthermore, requests with lower resource demand and lower scalability (resource efficiency) should also be executed earlier with less-demanding execution options (but slower) instead of high-demanding options to avoid blocking more resources for extended periods. The rest of this section discusses how to compute the priority f_i of request θ_i at time interval T by quantizing benefits, urgency, resource demand and scalability.

Algorithm 1 MEO VM Provisioning at interval T

```
1: function MEO-TSRA( )
2:     // collect requests and resource availability
3:     Θ = {θ_i | θ_i ∈ (New Arrival or Wait Queue)};
4:     Π = UpdateAvailCapacity();
5:     // prioritize user requests
6:     Q ← Prioritize(Θ);
7:     // allocate resources
8:     S = MEO-Greedy(Q, Π) or MEO-Adaptive(Q, Π);
9: end function
10: function Prioritize(Θ)
11:     for (θ_i ∈ Θ ) do
12:         calculate f_i; [Eqn(3)]
13:     end for
14:     Q ← Sort(Θ); // in descending order of requests' f_i
15: end function
```

For a request θ_i, its benefit is its bid b_i. Its urgency can be represented by $(d_i - t_0)$, where d_i is its deadline and t_0 is the beginning time of T. The resource demand and scalability of θ_i are calculated based on the demand and scalability of its execution options and available computing nodes.

Without loss of generality, consider the k'th execution option $E_{i,k}$ of θ_i and node $\mathbb{N}_p$. Let $y^r_{i,k}$ denote the total amount of $\mathbb{R}_r$-type resource demanded by $E_{i,k}$. Clearly, $y^r_{i,k} = w^r_{v_{i,k}} \cdot m_{i,k}$. If $E_{i,k}$ is allocated on node $\mathbb{N}_p$, the total percentage of resources used by $E_{i,k}$ on $\mathbb{N}_p$ over its whole execution, which is also its resource demand on $\mathbb{N}_p$, is then $t_{i,k} \cdot \sum_{\mathbb{R}_r \in \Gamma} \frac{y^r_{i,k}}{c^r_p}$.

For the scalability of $E_{i,k}$, we define it as the ratio of the reduced execution time over the increased amount of resources. More specially, For resource type $\mathbb{R}_r$, $E_{i,k}$'s scalability, $sl^r_{i,k}$ is then calculated as

$$sl^r_{i,k} = |\frac{t_{i,k-1} - t_{i,k}}{y^r_{i,k} - y^r_{i,k-1}}| \qquad (1)$$

Because an execution option's scalability is bounded by its least-scalable resource, the scalability of $E_{i,k}$, denoted by $sl_{i,k}$, is then $sl_{i,k} = \min_{\mathbb{R}_r \in \Gamma}(sl^r_{i,k})$.

With the benefit, urgency, resource demands and scalability, we can then compute the priority $f_{i,k,p}$ for execution option $E_{i,k}$ if it is allocated on node $\mathbb{N}_p$,

$$f_{i,k,p} = \frac{b_i}{sl_{i,k} \cdot (d_i - t_0) \cdot (t_{i,k} \cdot \sum_{\mathbb{R}_r \in \Gamma} \frac{y^r_{i,k}}{c^r_p})} \qquad (2)$$

In Equation (2), execution options with higher benefits, high urgency, lower scalability and less resource demands receive higher priorities. By considering all eligible computing nodes that have remaining resources available at time T for θ_i, we

Algorithm 2 MEO-Greedy VM Provisioning Scheme

```
1:  function MEO-Greedy(Q, Π)
2:      S ← Ø; // the set of selected requests is empty initially
3:      while (Q is not empty) do
4:          θ_i ← pop(Q); // pick job with highest priority
5:          // find a feasible execution option for θ_i
6:          while (E_i is not empty) do
7:              // pick the first or last exec opt
8:              E ← remove-first(E_i) or remove-last(E_i);
9:              if E meets deadline and has res. for E then
10:                 p_i = MAP(E, Π); // map E to nodes
11:                 Π = UpdateAvailCapacity();
12:                 S ← S ∪ ⟨θ_i, p_i⟩; // add θ_i to selected jobs
13:                 break; // θ_i done
14:             end if
15:             // E is not feasible, go on to check next opt
16:         end while
17:         if θ_i is not selected to run then
18:             if θ_i still has opts not missing deadline then
19:                 WaitQueue.add(θ_i);
20:             else
21:                 Discard θ_i;
22:             end if
23:         end if
24:     end while
25:     return S;
26: end function
```

define the overall MEO factor f_i (priority of θ_i) as the highest $f_{i,k,p}$ of all nodes and execution options,

$$f_i \quad = \quad \max_{\mathbb{N}_p \in \Pi} \{ f_{i,k,p} | \forall \mathbb{R}_r \in \Gamma, \forall k \in [1,j], y^r_{i,k} \leq c^r_p \} \quad (3)$$

f_i is used in Algorithm 1 line 12 for prioritization.

Algorithm 3 MEO-Adaptive VM Provisioning Scheme

```
1:  function MEO-Adaptive(Q, Π)
2:      S ← Alloc-First(Q, Π);//Get TSRA-First allocation
3:      S ← ScaleUp(S, Π);//Selectively scale-up jobs
4:      return S;
5:  end function
6:  function ScaleUp(S, Π)
7:      S2 ← S; // make a copy of the initial allocation
8:      // Repeatedly scale-up jobs if possible
9:      while (S2 is not empty) do
10:         S2 ← Rank-Scalability(S2);
11:         θ_i ← top(S2); // pick the most scalable job
12:         if (enough resource for θ_i to run next option) then
13:             Update θ_i with its next option in S and S2;
14:             Π = UpdateAvailCapacity();
15:         else
16:             pop(S2); // move on to next most scalable job
17:         end if
18:     end while
19:     return S;
20: end function
21: function Rank-Scalability(S)
22:     for θ_i ∈ S do
23:         calculate ScalabilityFactor_i for θ_i using Eqn(4);
24:     end for
25:     S ← Sort(S); // sort based scalability factor
26: end function
```

C. MEO-Greedy

MEO-Greedy is our first resource allocation heuristic scheme. Algorithm 2 gives the pseudo-code of this scheme. MEO-Greedy takes the prioritized jobs and available resources as input parameters. It then iterates over the requests from the highest priority to lowest priority (line 3 and 4) to allocate resources for them. For request θ_i, MEO-Greedy scans over its execution options one by one and greedily pick the first option that is feasible to be allocated (line 6-16). When scanning the options, MEO-Greedy may either process the options from the first (slowest and least-demanding) option to the last (fastest and most-demanding) option (which is called MEO-Greedy-First), or it may go from the last to the first (which is called MEO-Greedy-Last). Intuitively, MEO-Greedy-First picks the option with the highest resource efficiency, while MEO-Greedy-Last picks the option with the highest chance of increasing overall system utilization. If the current option E is feasible (line 9), it will be allocated and mapped to a node to execute (line 10-13). We use a *Euclidean Distance (ED)* mapping strategy presented in our recent work [3]. If none of the options of θ_i can be allocated due to resource limitation, θ_i will be added to wait queue (line 19). If all of θ_i's options miss its deadline, θ_i misses deadline and is discarded (line 21).

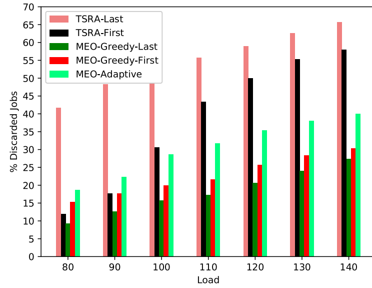
D. MEO-Adaptive

MEO-Adaptive is our second resource allocation scheme, the pseudo-code of which is given in Algorithm 3. MEO-Adaptive first allocates resources to the requests using their first (slowest) options (line 2). This allocation step is similar to the MEO-Greedy algorithm except it always uses the first option. Due to space limitation, the exact algorithm of this first-option allocation is not included in the paper.

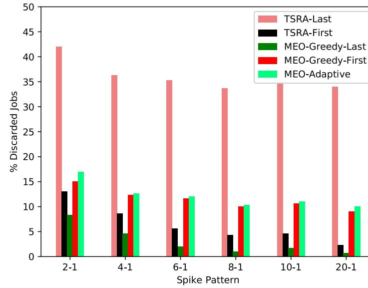
If there are resources available after the above allocation, MEO-Adaptive tries to scale-up the requests until most of the resources are allocated (line 3). To scale-up, MEO-Adaptive examines the next (faster and more demanding) options of all requests to pick the request with the most scalable next-option (line 10-11) and scale-up this request with this option (line 13-14). MEO-Adaptive repeats this scale-up operation until there is not enough resource left to scale-up any requests (line 9). Intuitively, MEO-Adaptive aims at maximizing resource utilization with the most efficient options.

We use Equation (4) to evaluate the scalability of execution options. For request θ_i, let E_k be the current option used in allocation. Let E_{k+1} be the next option that is faster and more demanding than E_{k+1}. The scalability of E_{k+1} can then be computed by dividing the reduced execution time over increase resources, as shown Equation (4). The scalability factors acquired with Equation (4) are used to rank all options of all requests (line 21-26).

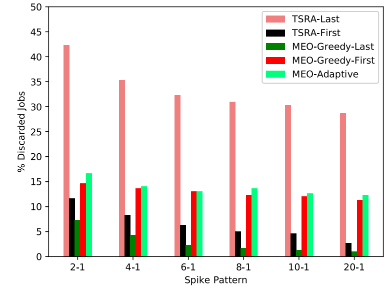
$$\mathrm{Scalabilityfactor_i} \quad = \quad \sqrt{\sum_{\mathbb{R}_r \in \Gamma} \left(\frac{t_{i,k} - t_{i,k+1}}{y^r_{i,k+1} - y^r_{i,k}} \right)^2} \quad (4)$$



a. different average system loads

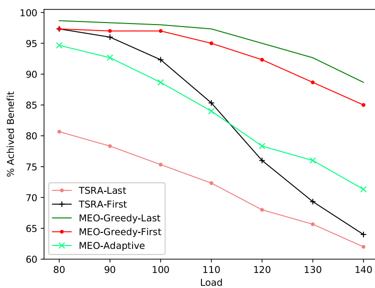


b. 20% loads with spikes of 180%

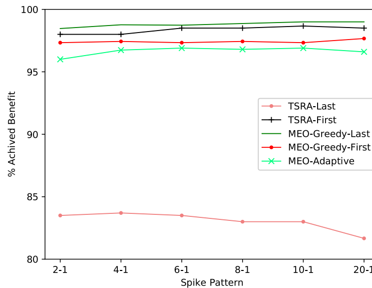


c. 30% loads with spikes of 170%

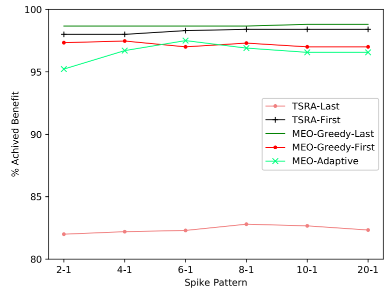
Fig. 2: Discarded requests for trace data of applications from NAS benchmark suite.



a. different average system loads



b. 20% loads with spikes of 180%



c. 30% loads with spikes of 170%

Fig. 3: Achieved benefit for requests from the trace data of applications from NAS benchmark suite.

V. Evaluations and Discussions

We have evaluated the performance of the proposed MEO-aware VM provisioning schemes through extensive simulations. In addition to the proposed *MEO-Greedy-First*, *MEO-Greedy-Last* and *MEO-Adaptive* schemes, for comparison, we also evaluated the TSRA scheme in our prior work [2] by considering only the first and last execution options of requests, which are denoted as *TSRA-First* and *TSRA-Last*, respectively. In what follows, we first present the settings of the simulations.

A. Simulation Settings

Although the proposed VM provisioning schemes can handle multiple types of resources, we have focused on only two types of resources (i.e., CPU cores and memory) in the evaluations. We considered a cloud system with 16 heterogeneous computing nodes, where two nodes have the capacity of $(16c, 64GB)$, two nodes of $(8c, 32GB)$, four nodes of $(4c, 8GB)$ and eight nodes of $(2c, 4GB)$, which represent a mixture of typical computing nodes.

The same as in our prior work [2], eight (8) different types of VMs are considered with their configurations being shown in Table II. The core and memory configurations for these VMs are actually derived from the widely deployed Amazon EC2 instances. In addition, the table shows the range of bids per time unit for each VM when it is requested to run any application. In the evaluations, we consider both applications from NAS Parallel Benchmarks suite and synthetic applications with randomly generated execution times.

TABLE II: VM configurations and prices

VM	CPUs	Memory(GB)	bid/time unit
c4.large	2	3.75	[0.05, 0.15]
c4.xlarge	4	7.5	[0.15, 0.25]
c4.2xlarge	8	15	[0.35, 0.45]
r4.xlarge	4	30.5	[0.22, 0.32]
r4.2xlarge	8	61	[0.48, 0.58]
m4.xlarge	4	16	[0.15, 0.25]
m4.2xlarge	8	32	[0.35, 0.45]
m4.4xlarge	16	64	[0.75, 0.85]

B. Performance for Requests of NAS Benchmark Applications

For the applications in the NAS Benchmarks suite, we have run them in both OMP and MPI versions on different numbers and types of VMs in the OpenStack environment installed on a cluster of 10 computing nodes. The detailed execution times for these applications running on different VMs can be found in the technical report of [3]. Based on the collected trace data regarding the applications' execution times on different VMs, the user requests to utilize the computing resources of the considered cloud system can be generated.

Here, for each request, its application is first randomly selected from the benchmark suites. Then, from the collected trace data, the available execution options (the number and type of VMs as well as the application's execution times)

are composed. For each option, the bid to utilize the VMs can be randomly generated using the bid range as shown in Table II and the bid of the request is set as the average bid of the available options. A request's arrival time is randomly generated and its deadline is set as 2 to 4 times of the longest execution time of all execution options from its arrival time.

We consider two different workload patterns and Figure 2 shows the percentage of discard requests under all VM provisioning schemes. First, the average system loads from 80% to 140% are evaluated. Here, for a given average system load (e.g., 100%), the workload in each interval is randomly set in the range of $+/-40\%$ (e.g., 60% to 140%). Considering the workload imposed by their first execution options, requests are generated until the accumulated workload reaches the target value. Each data point is the average of 10 trials of simulation and 1000 intervals (of 10 minutes) are evaluated for each trial.

As Figure 2a shows, our MEO-aware VM provisioning schemes perform significantly better than *TSRA-Last*, which uses only the last and least-efficient execution option of the requests with considerably increased resource demands. Although *TSRA-First* performs relatively better when the load is low, its performance deteriorates quickly for higher loads. The reason is that the adopted first options under *TSRA-First* introduce relatively less workload where resources may not be fully utilized at low loads and most requests can be served. However, at higher system loads, the single execution option limits the opportunities to effectively utilize available resources and leads to more discarded requests.

It turns out that *MEO-Greedy-Last* performs constantly the best. The reason is that, by starting with the last (and most-demanding) option of the requests, it can actually strike a good balance between resource efficiency and system utilization, where more requests can be served in time. Compared to *TSRA-First* and *TSRA-Last*, up to 30% and 39% more requests, respectively, can be served under *MEO-Greedy-Last*. For *MEO-Greedy-First*, it emphasizes more on resource efficiency than overall utilization and performs slightly worse than *MEO-Greedy-Last*. Moreover, *MEO-Adaptive* turns to have the worst performance among the MEO-aware schemes. This is probably due to the irregular execution options for the benchmark applications obtained from the trace data, which have quite large differences in their resource demands and limit *MEO-Adaptive*'s ability to gradually scale-up resource usages.

Second, we considered seasonal workloads with spikes, which are similar to some web and HPC applications. In particular, we consider $X:1$ patterns that have X intervals of low load interleaved with one interval of high load. The results for 20% vs. 180% and 30% vs. 170% workloads are shown in Figures 2bc, respectively. As X increases, there are more low-load intervals that provide better chances to serve the requests in high-load intervals in time and thus fewer requests are discarded.

Again, *MEO-Greedy-Last* performs the best due to its good balance of resource efficiency and system utilization, while *TSRA-Last* performs the worst due to its lowest resource efficiency. However, *MEO-Greedy-First* and *MEO-Adaptive* can perform worse than *TSRA-First*. The reason is that both MEO schemes try to exploit options with more resources for some requests, which drastically reduce resource efficiency due to the irregularity of the options and block considerably more resource from being allocated to future requests.

Figure 3 gives the corresponding achieved benefits (as the ratio of accumulated bids for the requests served in time over total bid for all requests) of all schemes for the benchmark applications under different workloads. With the similar reasonings for discarded requests, *MEO-Greedy-Last* performs the best among all schemes and up to 27% better than *TSRA-Last*. However, for spiky workloads, because the differences among discarded requests are relatively small where most discarded requests have low bids, the achieve benefits under the schemes (except *TSRA-Last*) are very close.

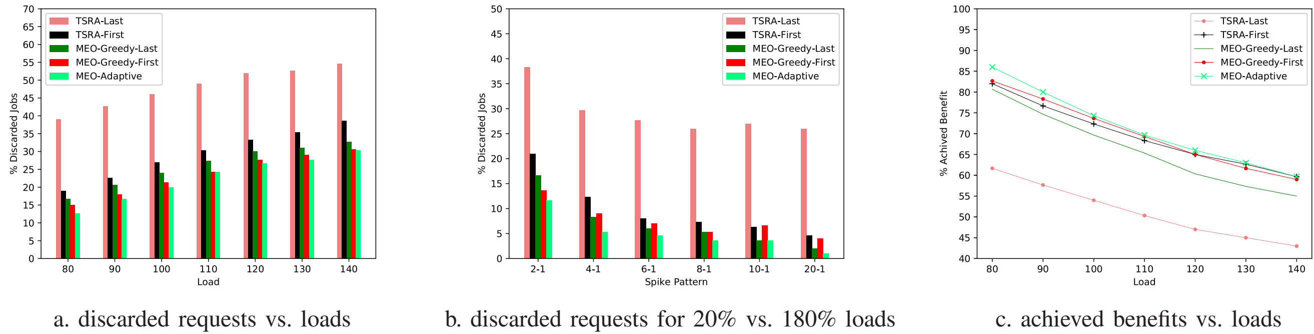
C. Performance for Synthetic Requests

To eliminate the impacts of irregular resource demands in execution options of benchmark applications, synthetic requests with more regular execution options are considered. More specifically, for each synthetic request, we first arbitrarily pick a VM from Table II with the execution time being randomly chosen between 5 to 200 minutes as its first execution option. Then three (3) more execution options are generated by increasing the resource demands gradually. For two consecutive options $E_{i,k}$ and $E_{i,k+1}$, the resource demand is usually increased by a ratio of x, where x is randomly picked between 1 and 3. The execution time reduces by a factor between 1 and x to reflect the non-linear scalability.

Figure 4 shows the performance of all schemes for synthetic requests with different loads. In general, the results show the similar trends for the schemes when system load changes. However, as the execution options of requests become more regular in resource demands, all MEO-aware schemes outperform TSRA, where *MEO-Adaptive* performs the best as it can adaptively choose the most resource-efficient execution options for requests and improve overall system utilization.

VI. Conclusions

In this work, we studied the VM provisioning problem for time-sensitive requests with multiple execution options (MEO) on resource-constrained clouds. We demonstrated that MEOs allow more flexible resource allocation which can considerably increase system utilization, deadline satisfaction and overall system benefit. To realize the full potential of MEO, we present two MEO-aware VM provisioning schemes, MEO-Greedy and MEO-Adaptive, that are carefully designed to balance overall system utilization, resource efficiency, request deadlines and benefits. We evaluated the proposed MEO-aware schemes on various workloads with both benchmark requests and synthetic requests. The results show that our MEO-aware algorithms outperform the state-of-the-art single-execution-option schemes by serving up to 38% more requests and achieving up to 27% more benefits.



a. discarded requests vs. loads b. discarded requests for 20% vs. 180% loads c. achieved benefits vs. loads

Fig. 4: Performance of the VM provisioning schemes with synthetic requests

References

[1] TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] R. Begam, W. Wang, and D. Zhu. Timer-cloud: Time-sensitive vm provisioning in resource-constrained clouds. *IEEE Trans. on Cloud Comput.*, 2017.

[3] R. Begam, W. Wang, and D. Zhu. Virtual machine provisioning for applications with multiple deadlines in resource-constrained clouds. In *Proc. of Int'l Conf. on High Performance Comput. and Commun.*, 2017.

[4] M. Bjorkqvist, L. Y. Chen, and W. Binder. Opportunistic service provisioning in the cloud. In *Int'l Conf. on Cloud Comput.*, 2012.

[5] R. N. Calheiros and R. Buyya. Cost-effective provisioning and scheduling of deadline-constrained applications in hybrid clouds. In *Proc. of the 13th Int'l Conf. on Web Inf. Syst. Eng.*, 2012.

[6] R. N Calheiros and R. Buyya. Meeting deadlines of scientific workflows in public clouds with tasks replication. *IEEE Trans. on Parallel and Distrib. Syst.*, 25(7):1787–1796, 2014.

[7] H. Chetto and M. Chetto. Some results of the earliest deadline scheduling algorithm. *IEEE Trans. on Softw. Eng.*, 15(10):1261, 1989.

[8] R. V. d. Bossche, K. Vanmechelen, and J. Broeckhove. Online cost-efficient scheduling of deadline-constrained workloads on hybrid clouds. *Future Generation Computer Systems*, 29(4):973 – 985, 2013.

[9] L. Dagum and R. Menon. Openmp: An industry-standard api for shared-memory programming. *IEEE Comput. Sci. Eng.*, 5(1):46–55, 1998.

[10] A. Deldari, M. Naghibzadeh, and S. Abrishami. Cca: a deadline-constrained workflow scheduling algorithm for multicore resources on the cloud. *The journal of Supercomputing*, 73(2):756–781, 2017.

[11] G. Feng, S. Garg, R. Buyya, and W. Li. Revenue maximization using adaptive resource provisioning in cloud computing environments. In *Proc. of Int'l Conf. on Grid Comput.*, 2012.

[12] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proc. of the 11th European PVM/MPI Users' Group Meeting*, 2004.

[13] M. R Garey and D. S Johnson. A guide to the theory of np-completeness. *WH Freemann, New York*, 70, 1979.

[14] Y. Guo, P. Lama, J. Rao, and X. Zhou. V-cache: Towards flexible resource provisioning for multi-tier applications in IaaS clouds. In *Proc. of the 27th IEEE Int'l Symposium on Parallel & Distrib. Process.*, 2013.

[15] K. Halder, U. Bellur, and P. Kulkarni. Risk aware provisioning and resource aggregation based consolidation of virtual machines. In *Proc. of the 5th IEEE Int'l Conf. on Cloud Comput.*, 2012.

[16] S. He, L. Guo, Y. Guo, C. Wu, M. Ghanem, and R. Han. Elastic application container: A lightweight approach for cloud resource provisioning. In *Proc. of Int'l Conf. on Advanced Inf. Netw. and Appl.*, 2012.

[17] W. Iqbal, M. N Dailey, D. Carrera, and P. Janecek. Adaptive resource provisioning for read intensive multi-tier applications in the cloud. *Future Generation Computer Systems*, 27(6):871–879, 2011.

[18] B. Javadi, J. Abawajy, and R. O Sinnott. Hybrid cloud resource provisioning policy in the presence of resource failures. In *Proc. of the 4th IEEE Int'l Conf. on Cloud Comput. Technol. and Sci.*, 2012.

[19] R. Jeyarani, N. Nagaveni, and R. Vasanth Ram. Design and implementation of adaptive power-aware virtual machine provisioner (apa-vmp) using swarm intelligence. *Future Generation Computer Systems*, 28(5):811–821, 2012.

[20] S. Kim and Y. Kim. Application-specific cloud provisioning model using job profiles analysis. In *Proc. of IEEE HPCC*, 2012.

[21] G. Le, K. Xu, and J. Song. Dynamic resource provisioning and scheduling with deadline constraint in elastic cloud. In *Proc. of Int'l Conf. on Service Sciences*, 2013.

[22] D. Li, C. Chen, J. Guan, Y. Zhang, J. Zhu, and R. Yu. Dcloud: deadline-aware resource allocation for cloud computing jobs. *IEEE Trans. on Parallel and Distrib. Syst.*, 27(8):2248–2260, 2016.

[23] N. Lim, S. Majumdar, and P. Ashwood-Smith. Mrcp-rm: A technique for resource allocation and scheduling of mapreduce jobs with deadlines. *IEEE Trans. on Parallel and Distrib. Syst.*, 28(5):1375–1389, May 2017.

[24] M.M. Nejad, L. Mashayekhy, and D. Grosu. Truthful greedy mechanisms for dynamic virtual machine provisioning and allocation in clouds. *IEEE Trans. on Parallel and Distrib. Syst.*, 26(2):594–603, Feb 2015.

[25] B. Nichols, D. Buttlar, and J. P. Farrell. *Pthreads Programming*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1996.

[26] Michael J. Quinn. *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill Education Group, 2003.

[27] S. Rizou and A. Polyviou. Towards value-based resource provisioning in the cloud. In *IEEE Conf. on Cloud Comput. Tech. & Sci.*, 2012.

[28] L. Shi, B. Butler, D. Botvich, and B. Jennings. Provisioning of requests for virtual machine sets with placement constraints in IaaS clouds. In *Proc. of IFIP/IEEE Int'l Symposium on Integr. Netw. Manag.*, 2013.

[29] A. N. Toosi, R. O. Sinnott, and R. Buyya. Resource provisioning for data-intensive applications with deadline constraints on hybrid clouds using aneka. *Future Generation Computer Systems*, 79:765 – 775, 2018.

[30] C. Vázquez, E. Huedo, R. S Montero, and I. M Llorente. On the use of clouds for grid resource provisioning. *Future Generation Computer Systems*, 27(5):600–605, 2011.

[31] C. Vecchiola, R. N Calheiros, D. Karunamoorthy, and R. Buyya. Deadline-driven provisioning of resources for scientific applications in hybrid clouds with aneka. *Future Generation Computer Systems*, 28(1):58–65, 2012.

[32] T. Voith, K. Oberle, and M. Stein. Quality of service provisioning for distributed data center inter-connectivity enabled by network virtualization. *Future Generation Computer Systems*, 28(3):554–562, 2012.

[33] W. Wang, B. Liang, and B. Li. Multi-resource fair allocation in heterogeneous cloud computing systems. *IEEE Trans. on Parallel and Distrib. Syst.*, 2014.

[34] F. Wuhib and R. Stadler. Distributed monitoring and resource management for large cloud environments. In *Proc. of IFIP/IEEE Int'l Symposium on Integr. Netw. Manag.*, 2011.

[35] S. Zaman and D. Grosu. Combinatorial auction-based mechanisms for vm provisioning and allocation in clouds. In *Proc. of Int'l Symposium on Cluster, Cloud and Grid Comput.*, 2012.

[36] Q. Zhu and G. Agrawal. Resource provisioning with budget constraints for adaptive applications in cloud environments. In *Proc. of Int'l Symposium on High Performance Distrib. Comput.* ACM, 2010.